Formal Safety Analysis in Industrial Practice

Ilyas Daskaya[1], Michaela Huhn[2], and Stefan Milius[1]

[1] Institut für Theoretische Informatik, Technische Universität Braunschweig
Braunschweig, Germany
{I.Daskaya|S.Milius}@tu-braunschweig.de
[2] Department of Informatics, Clausthal University of Technology
Clausthal-Zellerfeld, Germany
Michaela.Huhn@tu-clausthal.de

Abstract. We report on a comparative study on formal verification of two level crossing controllers that were developed using SCADE by a rail automation manufacturer. Deductive Cause-Consequence Analysis of Ortmeier *et al.* is applied for formal safety analysis and in addition, safety requirements are proven. Even with these medium size industrial case studies we observed intense complexity problems that could not be overcome by employing different heuristics like abstraction and compositional verification. In particular, we failed to prove a crucial liveness property within the SCADE framework stating that an unsafe state will not be persistent. We finally succeeded to prove this property by combining abstraction and model transformation from SCADE to UPPAAL timed automata. In addition, we found that the modeling style has a significant impact on the complexity of the verification task.

Keywords: model-based development, SCADE, Deductive Cause-Consequence Analysis

1 Introduction

A key issue in the development of safety-critical systems is safety analysis, i. e., a thorough analysis how components may fail and cause a system hazard. From the safety analysis, the safety requirements for components are derived and the design is proven correct w.r.t. functional safety in the verification and validation phase.

For developing software for safety-critical systems, formal methods are considered an adequate means because they provide correctness results at a level of strict mathematical rigor. Standards like the IEC 61508 part 3 [15] and its railway-specific derivative CENELEC 50128 [6] highly recommend them for the software development according to higher safety integrity levels (SIL). Nevertheless, seamless usage of formal methods is still a future challenge for industries. Frequently heard arguments trying to explain industries' indecision towards the proliferation of formal methods are a lack of integration and coverage in the development process and poor scalability for larger designs.

In this paper we report on our experiences on a seamless, tool supported, formal approach that covers design, safety analysis and formal verification: We compare two medium-sized design variants of a level crossing controller. Both were developed using

the SCADE Suite[3] by a manufacturer for rail automation equipment. The two models are different in their modeling style, namely one is state-based and built based on safe state machines [4], whereas the other one is strictly data-flow oriented. Both implement the same functionality and have exactly the same interfaces, but the models were originally thought as a comparative basis to evaluate model qualities relevant for application developers such as understandability, maintainability etc. This was a purely design-oriented comparison that we have expanded to safety analysis and formal verification.

So the starting point for our investigation is the question whether the modeling style has an impact on formal safety analysis and verification. By using the built-in state-of-the-art SAT-based model checker of SCADE we also strive for a good showcase for seamless formal support of the safety process. A precondition for an authentic showcase is to take the original design models without tailoring them beforehand to the needs of some particular formal analysis technique at hand.

We apply *Deductive Cause-Consequence Analysis* (DCCA) by Ortmeier et al. [19] as a formal generalization of the well accepted safety analysis techniques FMEA (*Failure Mode and Effect Analysis*) and FTA (*Fault Tree Analysis*). Moreover, we are able to formally verify a number of safety requirements, i.e. functional correctness properties imposed by the safety analysis. However, we have to cope with severe complexity problems even with these medium sized designs that prevent us from completing both, the DCCA and the verification of safety requirements. We try several abstraction techniques to reduce complexity of the verification problems within the SCADE Design Verifier, but without success. We illustrate our attempts on data abstraction, cone of influence, and symmetry reduction as well as decomposition using the example of a liveness property that arises from the safety requirements. Finally, we transform the state-based model into UPPAAL timed automata[4] and succeed with the verification of that liveness property easily. What we technically realize as a model transformation between two formal frameworks, namely SCADE and UPPAAL, is methodically an *abstraction of time*: In the SCADE model time is handled in multiple steps in each of which an external timer is compared to internal variables modeling timeouts. In contrast, the real-time zones of UPPAAL only take *one* transition to the next relevant point in time. Hence the resulting state space is significantly smaller.

Overall, the state-based model of the level crossing control can be considered slightly better suited for formal analysis. However, to be able to generalize this result, a clear and *application-oriented* notion of model elements with major impact on the verification complexity has to be developed.

The paper is structured as follows: In Sec. 2 we recall safety analysis using FMEA and DCCA as a formal approach to it and SCADE suite as a model-based development environment featuring formal verification. Liveness analysis, a number of heuristics we have tried in order to deal with the intrinsic complexity problems, and, notably, the transformation of SCADE models to UPPAAL timed automata are described in Sec. 3. In Sec. 4, safety analysis and verification results of the two model variants of a level crossing controller are presented in detail. Sec. 5 concludes with lessons learned.

---

[3] SCADE is a product of Esterel Technologies, see `www.esterel-technologies.com`.

[4] UPPAAL is an integrated tool for modeling and verification of real-time systems, see `www.uppaal.com`.

2 Safety Analysis and DCCA

The development of safety-critical systems and their software is regulated by standards. In the railway domain the CENELEC standards EN 50126, 50128, and 50129 [9, 6, 10] apply. Since software is intangible, it is commonly agreed that system failures caused by software malfunction stem from systematic errors that are introduced during the software development and are not recognized in the safety process. Consequently, software safety engineering aims at (1) a complete and consistent specification of functional and safety aspects for a software component, (2) the correct implementation of the specification and (3) providing evidence that the safety objectives are met.

2.1 Safety Analysis

In the architectural design phase, the intended functionality is modularized. For safety-critical systems, a *hazard analysis* is performed to identify potential failures, hazards, and the causal chains between them. Classical inductive methods for hazard analysis are *Failure Mode and Effect Analysis* (FMEA) and its extension *Failure Mode, Effects and Criticality Analysis* as standardized in IEC 60812 [14]: FMEA classifies failures according to the severity of their effects, the occurrence frequency, and the detection rate. Starting from a definition of the system and its boundaries, a functional viewpoint is taken and each subfunction is analyzed with respect to potential fault modes. For functionality implemented in software, fault identification is supported by generic fault types like omission, commission, untimely reaction and value fault [18]. Local effects can be directly deduced from the component faults. In order to determine the system level effects, fault propagation is analyzed based on the functional architecture by using a formal deduction method (see Section 2.4). A fault is called *critical* if it has severe effects and an unacceptably high occurrence frequency. Safety measures are taken to eliminate the faults or at least mitigate the effects, to decrease their occurrence probability, or to actively detect them. Findings and actions to be taken are usually summarized in an FMEA table. Often a Fault Tree Analysis (FTA) is conducted to complement FMEA. FTA proceeds deductively by starting from potential hazards and then investigates which combination of faults or operational modes in the components may cause them.

FMEA is a structured but semi-formal technique. In an iterative design process, an FMEA is conducted and refined with each iteration cycle or change in the design or operational constraints. As FMEA is an inductive method, common cause faults and their effects are analyzed in a separate step.

2.2 Safety-Oriented Software Design

For the development of safety-related software in the rail domain, the standard CENELEC 50128 [6] prescribes the activities for design, validation, and verification with their input and output artifacts. As faults due to software are traced back to systematic development errors, the standard recommends measures that are considered appropriate to avoid such errors or to reveal them in a validation or verification step. Hence, development activities have to be performed with an adequate degree of rigor such that

functional correctness and fulfillment of safety requirements can be proven with substantial evidence.

The SCADE tool suite (Safety-Critical Application Development Environment) is a model-based development framework for safety-critical software that supports certification due to EN 50128 up to SIL 3 and 4. The SCADE modeling language is a synchronous and dataflow-oriented language based on LUSTRE [12], and was extended by safe state machines [4]. SCADE provides certified automated code generation and immediate simulation of the model. It supports testing, in particular model coverage is recorded, and the SCADE Design Verifier (SCADE DV) allows for SAT-based formal verification[5].

Here, SCADE was chosen as design framework for the level crossing control (LC) from our industrial partner. The reason was the seamless design flow from the requirements via the model executing on a hardware abstraction layer (HAL) to the C code generated for the real target processor. Back then, validation, verification and safety assessment were performed with traditional methods. With a broader usage of SCADE, different modeling styles came up and also the desire to benefit from the Design Verifier.

2.3 Formal Verification Using SCADE DV

The behavior of a SCADE design model can be given as a transition system on which SAT-based model checking can be performed in order to verify reachability properties[6], see [1].

In contrast to other approaches, the SCADE DV does not offer a temporal logic but properties have to be modeled as *synchronous observers* using the same language operators as for the design. Nevertheless, we will use CTL as a succinct notation for reachability properties to be verified, but the reader should keep in mind that they are encoded as observer nodes at the SCADE level (cf. Figures 1 and 5). Notice, however, that more general temporal properties, notably unbounded liveness, cannot be automatically verified using this SAT-based approach.

2.4 Deductive Cause Consequence Analysis

An important step in safety analysis concerns determining the causal relationship between component fault modes and hazards. There are various techniques that exploit formal methods, notably model checking, for identifying the desired cause-consequence relation [1, 16, 5]. Here we consider DCCA by Ortmeier et al. [19, 11], which is a formal approach to safety analysis that generalizes FTA and FMEA. In DCCA, a hazard H is specified as a state predicate. Primary component fault modes are modeled by adding simple fault automata that are triggered by a Boolean input indicating the occurrence of this particular fault. The immediate effect of a fault on a component is specified in a so-called fault node within the component model. The hazard is implemented as an

[5] SCADE uses the SAT solver developed by Prover Technologies, see `www.prover.com`

[6] In the area of model checking, reachability properties are often called *safety properties*, because they express that the system always stays in a good or "safe" state. In contrast, for us a safety property is any constraint to ensure dependable behaviour.

observer node that takes signals from the system and evaluates them according to the *negation* of the hazard predicate H, i.e., returning false whenever the hazard occurs. Figure 1 gives a schematic picture of the integration of DCCA with SCADE.

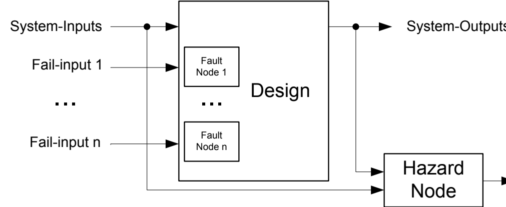

Fig. 1. Integration of DCCA with SCADE

A core notion of FTA is that of a *critical cut set* of faults for a hazard, i.e. a subset of primary faults for which some operational condition and occurrence sequence exist such that the hazard occurs without further influence. This was formalized in DCCA to the notion of a *critical set*. According to [11], critical sets can be formalized for the SCADE semantics as follows:

Definition 2.1. *Let Γ be a subset of the set of primary component faults Δ of a system* Sys. $\mathsf{Sys}\rangle_{\overline{\Gamma}}$ *denotes that behavior of* Sys *where no fault from $\Delta \setminus \Gamma$ ever occurs. Now Γ is called* critical *for a hazard described by the state predicate H iff* $\mathsf{Sys}\rangle_{\overline{\Gamma}} \not\models \mathbf{AG}\,\neg H$. *A critical set Γ is called* minimal *iff no proper subset of Γ is critical.*

If the analysis reveals a singleton as minimal critical set, this indicates that there exists a single-point-of-failure in the system, which has to be eliminated by further safety measures. A DCCA is called *complete* iff all minimal critical sets have been identified.

Theorem 2.2 (Minimal-Critical-Set Theorem [19, 11]). *If a complete DCCA has been conducted for a hazard H then for each minimal critical set, preventing* one *fault from occurrence will prevent the hazard H.*

In order to determine the minimal critical sets we use an iterative approach similar to [1]: Suppose that Δ is the set of all component fault modes identified during the FMEA and that H is a hazard. We iterate over all subsets $\Gamma \subseteq \Delta$ starting with $\Gamma = \emptyset$ and increasing Γ stepwise (singletons, doubletons etc.). We use the SCADE DV to prove whether Γ is critical for H: all trigger inputs for fault modes in Γ become system inputs, and the inputs of all other fault mode nodes are constantly false. Whenever SCADE DV returns a counterexample, i. e. a situation in which the hazard node expressing $\neg H$ is false, we have identified a minimal critical set Γ, and due to the monotonicity of criticality no super set of Γ needs to be considered.

3 Liveness and Abstraction Techniques

In this section we describe the general steps we took in our case study. Here we assume that we are given a design model of the system Sys in a formalism providing support for formal verification. Our steps were as follows:

(1) Given the system model we performed a safety analysis identifying hazards and component fault modes, cf. Section 2.1.

(2) We used DCCA to determine the cause-consequence relationship between component fault modes and hazards. For this we use the SCADE DV to obtain minimal critical sets of fault modes, cf. Section 2.4.

(3) We performed an analysis of a liveness property arising from the safety requirements of the system.

(4) We applied several strategies in order to deal with complexity issues arising in connection with the model checking performed in steps (2) and (3).

We already explained methodological details of steps (1) and (2) in Section 2. In the remainder of this section we will describe steps (3) and (4) in more detail.

3.1 Liveness Analysis

This step of our case study concerns the analysis of a safety related liveness property derived from a system requirement, see Section 4.3 for the concrete formulation. Roughly, this requirement states that a certain non-safe state of the system may occur, but this state must not be permanent, and it must be left within a certain time interval that depends on the system's configuration. Indeed, our analysis in step (2) revealed that this non-safe state of the system will occur under a certain fault mode. This is still in accordance with the requirement. However, it has to be verified that the system will leave this non-safe state in time.

If φ is a propositional formula describing the non-safe state then this property might be formulated in a temporal logic like CTL as $\mathsf{AG}(\varphi \to \mathsf{AF}\neg\varphi)$. However, SCADE does not provide a direct way to express a liveness property with an unbounded quantifier like AF. A concrete time limit – and hence a bound for the number of cycles – is not specified, but it is individually set for each configuration.

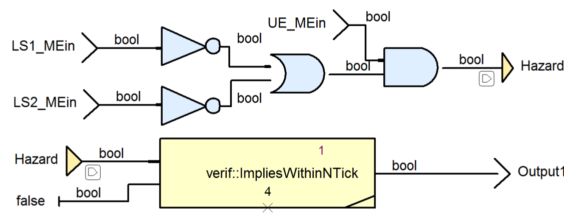

Fig. 2. Observer node for the liveness analysis with $n = 4$.

Thus, we explore the model and try to establish a lower bound on the number n of cycles after which the non-safe state is left: It is clear that once in the non-safe state the system will remain there for at least one cycle, and so we verify whether the non-safe state is left after $n = 2, 3, \ldots$. This can be modeled as a SCADE observer by using the operator ImpliesWithinNTicks from the design verifier library, see Figure 2 for $n = 4$.

3.2 Dealing with Complexity within SCADE DV

Unfortunately, we failed to complete our analysis just using SCADE DV even with small numbers n due to complexity problems, see Section 4 for details. We will now mention the strategies we have applied in order to deal with this problem.

Abstraction. First we generated an abstract version of the design model in SCADE. For this we applied three different techniques: data abstraction, cone of influence reduction and symmetry reduction (see e. g. [7] for an introduction) – the way in which these are applied is detailed in Section 4. However, even for the abstracted model the liveness verification could not be completed using SCADE DV.

Compositional verification. In a further attempt to cope with complexity of the liveness analysis we manually broke down the given liveness property into proof obligations for each component of the model. We then argued that the liveness property holds for the whole model if the components satisfy their respective proof obligations. This divide-and-conquer strategy allowed us to complete the verification for all but one model component with the SCADE DV.

3.3 Model Transformation to UPPAAL

In order to complete the analysis for this remaining system component we decided to transform this component to another formalism. Manual inspection of the component revealed that its behavior is to a large extent governed by 14 timers that are external to the SCADE model and are provided from the runtime environment in form of an integer input. For this reason we consider it promising to transform that component to a modeling formalism that also takes time into account. We have therefore chosen timed automata [3] with UPPAAL as modeling and analysis tool.

We will briefly describe the principles of the model transformation: We assume that the given SCADE node comes as a *flat* safe state machine. That means that there are no hierarchical states and no memory within states (such as fby operators). Outputs are assigned within the states. We also assume that all transitions in the safe state machine are *strong*, i. e., in a cycle where the transition guard holds the target state of the transition is active (see [4]). These restrictions hold for the model in our case study. For other models, preprocessing steps like flattening of hierarchical states have to be applied.

In contrast to SCADE's deterministic behavior, UPPAAL models may also behave non-deterministically. Thus, the transformation should ensure the deterministic behavior of the model in UPPAAL. The following points are important:

Activation conditions and output variables. The activation condition of a transition in SCADE gets mapped to the guard of the corresponding edge in UPPAAL. If an output flow of a SCADE node changes when taking a transition, we add the corresponding assignment along the translated transition in UPPAAL.

Firing transitions. When a transition guard holds, the corresponding transition in the translated UPPAAL model must fire immediately to faithfully reflect the transition behavior in the SCADE model. For this purpose we use UPPAAL urgent channels. Each translated transition synchronizes on an urgent channel that can always be activated so that no delay occurs along the transition, see Figure 8, where this urgent channel is go2.

Transition priorities. We must make sure no two transition guards are `true` simultaneously. SCADE uses explicit transition priorities to prevent this. Suppose that we have two transitions with the same source state and with guards φ and ψ, respectively, such that the φ transition has higher priority. Then in the transformed UPPAAL model the first transition has guard φ and the second one the guard $\psi \wedge \neg\varphi$.

Timers. As already mentioned, our SCADE model is partly governed by external timers that are started by certain model outputs and that trigger transitions. In the translated UPPAAL model these timers are explicitly modeled as shown in Figure 3. The timeouts are taken from the SCADE model. The edge between $S0$ and $S1$ will reset the clock variable x. During the urgent state $S1$ clock x does not progress. So any output assignment that may happen together with the start of the timer will be performed as a variable assignment at the edge from $S0$ to $S1$. The invariant $x \leq 10$ makes sure that the state $S1$ is left before x reaches 10. Upon timeout ($x == 10$) we progress to $S3$ and perform any output assignment associated with the timeout along the edge from $S2$ to $S3$.

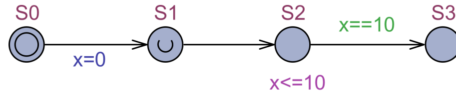

Fig. 3. Modelling a timeout in UPPAAL.

This part of the model abstraction realizes the time abstraction mentioned in the introduction. A situation where the SCADE model is waiting for a timeout, i.e., the integer input corresponding to system time increases for a (possibly large) number of cycles where no reaction of the model happens and no model output changes, corresponds to only one transition in the transformed UPPAAL model. This fact leads to a significant reduction in the size of the state space of the transformed model, and we believe this makes formal verification feasible.

3.4 Correctness of the Model Transformation

Although the transformation from SCADE to UPPAAL can be automated in principle, we manually performed it in our case study. We also did not provide a formal proof of the semantic correctness of our transformation. Both tasks were out of the scope of our current project, and so we leave them for future work.

However, in order to establish confidence in the correctness of our transformation we followed an approach based on testing that we will now describe. From the requirements specification we created a test suite. By using the SCADE model test coverage facility this test suite was shown to yield 90% decision coverage of the original SCADE model. The same test suite was used on the translated UPPAAL model. For each test case we recorded the traces (i.e., the list of input and output values in each step) of the simulation of both models.

Now in order to establish the equivalence of the two models we need an appropriate notion of equivalence for the traces. Due to the different timing concepts of the two underlying formalisms, the traces of the SCADE model do not correspond one-to-one to the traces of the translated UPPAAL model. Instead we use a version of *stuttering equivalence*, see e. g. [17, 7]. We write $(\vec{v}_1, \ldots \vec{v}_n)$ for a trace of length n produced by a model simulation running some test case. We say that this trace is *stuttering equivalent*

to some other trace $(\vec{w}_1, \ldots, \vec{w}_m)$ if there are sequences $1 = i_1 < i_2 \cdots < i_{k-1} < i_k = n$ and $1 = j_1 < j_2 \cdots < j_{k-1} < j_k = m$ with $k \leq n, m$ and such that

$$\vec{v}_{i_r} = \vec{v}_{i_r+1} = \cdots = \vec{v}_{i_{r+1}-1} = \vec{w}_{j_r} = \vec{w}_{j_r+1} = \cdots = \vec{w}_{j_{r+1}-1} \quad \text{and} \quad \vec{v}_n = \vec{w}_m,$$

where $1 \leq r \leq k-1$. In other words, two lists containing the same input/output values in the same order (but possibly repeating certain list elements a different number of times) are equivalent. Example: Suppose we have a model with one integer input and one Boolean output. Then the two lists $((1, \mathsf{true}), (1, \mathsf{true}), (42, \mathsf{false}), (42, \mathsf{true}))$ and $((1, \mathsf{true}), (42, \mathsf{false}), (42, \mathsf{false}), (42, \mathsf{true}))$ are equivalent.

With this notion of equivalence we compared traces of SCADE simulations of a test case with those of UPPAAL simulations of the same test case. We showed equivalence of the two models with respect to all test cases from our test suite.

4 Comparative Safety Analysis of Level Crossing Control

Now we present the results of our industrial case study. For the sake of brevity we omit some details; they can be found in [8]. Our formal safety analysis was performed comparatively for two SCADE models for the modular level crossing controller in [13]. The two SCADE models were developed by different developers with different implementation styles: the first model uses a design based on safe state machines (SSM) [4] and the second one uses data flow diagrams, which are essentially graphical respresentations of LUSTRE [12]. Whenever information has to be stored for the next execution cycle in the data-flow oriented model, this is done within local variables for which the value is kept using the fby-operator. Both models implement the same architecture with the following operators, which can be composed in order to obtain a level crossing control logic for a specific level crossing layout:

- *route controller* (LC_Route)
- *site controller* (LC_Site)
- *group controller* (LC_LS_Group)
- *time controller* (LC_Timer)

The route controller monitors the activation of the lights and barrier groups depending on the activation signal from a particular route, cf. Figure 4.

The site controller synchronizes all route controllers and group controllers and acts as a logical connector. The group controller controls the lights and barriers, which are grouped logically, using a hardware abstraction layer. Finally, the time controller monitors the time elapsed since the last activation of the LC until its complete deactivation. Depending on the level crossing layout, numbers of inputs, outputs and nodes in the models can vary. In our case, models are specified for controlling of a level crossing with 2 routes and 2 lights-barrier groups, see Figure 4. They have 18 boolean, 1 integer input variables, 5 boolean output variables and 14 constants. The models are composed of 5 nodes: 2 LC_Route, 2 LC_LS_Group, 1 LC_Site. Specifically, the first model has 12 states + 23 transitions for each LC_Route operator, 14 states + 23 transitions for each LC_Site operator, 27 states + 51 transitions for each LC_LS_Group operator. In both models, the LC_Timer operator has 2 states.

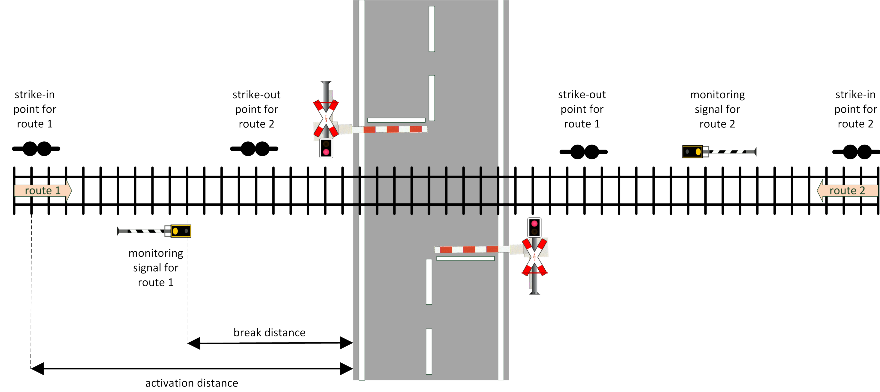

Fig. 4. Sample layout of a locally monitored level crossing

4.1 Application of FMEA

We briefly summarize the results of our FMEA, the details are in [8]. The analysis has revealed ten component fault modes (FM1 – FM10) and two hazards: (a) a train drives through a non-secured level crossing (LC) and (b) car drivers drive against closing/closed barriers. Hazard (a) can happen if the monitoring signal shows the LC to be safe while it is not (i. e., one of the light-barrier groups (LBG) is switched off). That means, if a monitoring signal is activated (*UE_MEin=true*), both LBGs must be switched-on (*LS1_MEin=true* and *LS2_MEin=true*). This state can be represented as a formula H in propositional logic as follows:

$$H = (UE_MEin \wedge \neg(LS1_MEin \wedge LS2_MEin)) \tag{1}$$

Figure 5 shows the SCADE observer node for the hazard H from (1).

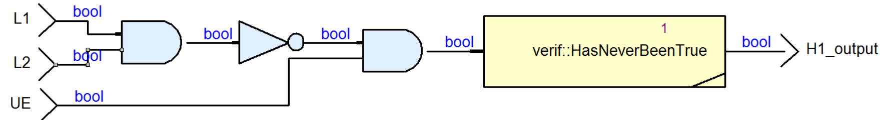

Fig. 5. SCADE model of the hazardous event H

4.2 Application of DCCA

To apply DCCA to our two models we first created an environment model in SCADE simulating stimuli from the hardware of a real level crossing system. We then extended the hardware model with the fault modes and their occurrence patterns from the FMEA.

Functional correctness. Recall from Section 2.4 that the first step of DCCA considers the empty set of fault modes. So we verify functional correctness of the models w.r.t. the safety requirement expressed by the hazard H. With SCADE DV, the proof took 15 seconds for the first model and 322 seconds for the second one. The difference

	State-based Model			Data-flow Model		
N=1	Valid	Critical	Time	Valid	Critical	Time
FM1		x	1		x	1
FM2	x		122	x		543
FM3	x		14	x		459
FM4	x		13	x		312
FM5	x		143	x		323
FM6	x		108	x		330
FM7	x		199	x		329
FM8	x		14	x		324
FM9	x		15	x		245
FM10	x		14	x		495
Average Time (sec)			64,3			336,1

Table 1. Singleton fault modes

originates from the different implementation styles. The SSM based design seems to have a smaller state space, hence a quicker proof is possible with the first model.

Single fault modes. We checked criticality of singleton fault mode sets Γ w.r.t. the hazard node H. Table 1 shows the results and proof execution times in seconds. Only FM1 (unwanted switch off of warning lights) is identified as critical for both models. The rest of the failure modes could be proven to be non-critical.

At most 2 fault modes. This step requires analysis of the fault mode sets Γ with $|\Gamma| = 2$. As FM1 is already critical, we only analyze sets Γ with FM1 $\notin \Gamma$. We assume that for each hardware type only one fault mode can occur at a time, and only a single hardware component of the same hardware type can fail at a time, e. g. only one of the barriers in a barrier group can fail. Finally, sensor failures can occur either as a false detection or a mis-detection, but not both.

Figure 6 and 7 present the analysis results for the first and second model, respectively. The cells marked with an "x" represent single fault modes and already have been treated in the previous step. The *Gray* colored combinations have not been analyzed due to the assumptions explained in the previous paragraph and since FM1 is critical. *Red* colored cells (marked by "C") represent the critical sets while *green* cells (marked by "V⟨time⟩") represent the non-critical fault modes and also indicate the proof durations in seconds. Critical sets were proven to be critical within 1 second. *White* colored combinations (marked by "U") could not be proven to be critical or non-critical in a reasonable time. We could not analyze all relevant 2-element fault mode combinations, hence, our DCCA for critical doubletons is not complete. Since three simultaneous failures are highly unlikely we terminated the analysis with this step.

	FM1	FM2	FM 3	FM4	FM5	FM6	FM7	FM8	FM9	FM10
FM1	x									
FM2		x								
FM3			x							
FM4		V149	V15	x						
FM5		V457	U		x					
FM6		U	U	C	V256	x				
FM7		U	U	C	V14		x			
FM8		U	U	U	U	U	U	x		
FM9		U	U	U	U	U	U		x	
FM10		U	U	U	U	U	U			x

Fig. 6. Results for the state-based model

	FM1	FM2	FM 3	FM4	FM5	FM6	FM7	FM8	FM9	FM10
FM1	x									
FM2		x								
FM3			x							
FM4		U	U	x						
FM5		U	U		x					
FM6		U	U	C	U	x				
FM7		U	U	C	U		x			
FM8		U	U	U	U	U	U	x		
FM9		U	U	U	U	U	U		x	
FM10		U	U	U	U	U	U			x

Fig. 7. Results for the data-flow model

To summarize, while our analysis remains incomplete we have identified three minimal critical sets: {FM1}, {FM4,FM6} and {FM4,FM7}; FM4 means that a barrier is stuck and FM6 and FM7 mean the corresponding sensor does not detect this.

4.3 Liveness analysis

We have shown that both models do have critical sets of fault modes. From a control engineering point of view this was expected to happen, and rather than requiring prevention of the hazard the corresponding safety requirement for the system reads as follows:

Requirement. *A state in which the monitoring equipment feedbacks its status as activated* (UE_MEin=true) *and at least one LBG is non-active* (LS1_MEin=false *or* LS2_MEin=false) *is non-safe. Such a state must not be permanent and has to be left independently from the input values as quickly as possible by deactivation of the monitoring signal.*

This requirement can be formalized as a liveness property in CTL as follows:

$$\mathsf{AG}(UE_MEin \wedge \neg(LS1_MEin \wedge LS2_MEin) \Rightarrow \mathsf{AF}(\neg UE_MEin)) \quad (2)$$

As explained in Section 3.3 we tried to use the SCADE DV with an observer as shown in Figure 2 to obtain a lower bound on the number n of cycles the system remains in the non-safe state corresponding to our hazard property H.

4.3.1 Liveness of the state-based model. For $n = 2$, SCADE DV delivered "falsifiable" as an answer within a few seconds, which means that the hazard may last for at least two cycles. For $n = 3$ it was not possible to prove the correctness of the system after 1 week of execution time as a result of the state explosion problem. To overcome this problem we used several strategies (see Section 3.3). Full details are in [8].

Data Abstraction. The only integer input of the models is used for synchronization of the controller with a global clock *T_System*. The value of *T_System* is used by 14 different timers for realizing the delays and timeouts. For the current verification, only 10 of them are relevant. Using SCADE assumptions we have put bounds on the possible values of *T_System*, thus reducing the complexity of the verification.

Cone of Influence Reduction. Only 7 inputs out of 19 of the model have an influence on the observer node. Some of the remaining inputs have constant values as they correspond to configuration settings. These inputs can be replaced by the constants. As a consequence some transition triggers are simplified or even become constantly false. As a result some states become unreachable and are removed as well.

Symmetry Reduction. Here we remove the identical operators and symmetric configurations in order to reduce the state space. Our initial configuration was a 2 Route+2 LBG level crossing. Obviously, a reduced model (1 Route+1 LBG) needs to satisfy the liveness property, too. If we can verify the validity, it can substantiate the claim that the full system is correct.

Unfortunately, even with the above three simplifications it was still impossible to verify the liveness property with the SCADE DV.

4.3.2 Compositional verification. With an operator level analysis, we identified that an occurrence of FM1 generates an *LS_MEin=false* (an LBG is not switched on) signal,

which is converted to *AN_MEin=false* (not all LBG's are switched on) signal through an *AND* gate by the *LC_LS_Group* operator. It feedbacks this status to the *LC_Site* operator, which updates an internal variable as *LS_MEin=false*. Finally, *LC_Route* receives this signal and signals the monitoring signal to switch-off. This is followed by a *UE_MEin=false* signal by a failure-free monitoring signal. This led us to decompose the liveness property for the whole model into properties for its components:

1. For *LC_LS_Group* operator: *LS_MEin=false* shall be followed by an *AN_MEin=false* signal; in CTL:

$$\mathsf{AG}((\neg LS_MEin \wedge AN_MEin) \Rightarrow \mathsf{AF}(\neg AN_MEin)). \tag{3}$$

2. For the *LC_Site* and *LC_Route* operators: *AN_MEin=false* shall be followed by an *UE_MEin=false* signal; in CTL:

$$\mathsf{AG}((UE_MEin \wedge \neg AN_MEin) \Rightarrow \mathsf{AF}(\neg UE_MEin)).$$

It can be easily seen that both parts together imply the main liveness property. Analysis of the second formula was successful. Thus, the operators *LC_Site* and *LC_Route* are verified. They were found to satisfy the liveness property within 6 execution cycles. However, the first part could not be proven in a reasonable time. This led us to consider the *LC_LS_Group* operator as the bottleneck for the complete analysis.

4.3.3 UPPAAL transformation. Using the model transformation described in Section 3 we transformed the *LC_LS_Group* operator to UPPAAL. The resulting UPPAAL model is a network of 5 timed automata. Four automata model the environment such as hardware behavior, switch-on and switch-off commands and the fault mode FM1. The *LBG automaton* (see Figure 8) models the behavior of the LC_LS_Group operator. It synchronizes with other UPPAAL automata over the urgent *go1* channel. This channel ensures progress and an immediate transition as soon as an edge's guard holds. For example, it can be seen that edge from *S00* to *S01* fires as soon as *AN_SEin* holds.

Verification with UPPAAL. UPPAAL uses a fragment of timed CTL (TCTL) [2]. Like TCTL, the query language consists of path formulæ and state formulæ [20]. In our case the liveness property (3) for the *LC_LS_Group* operator will be rewritten as:

```
(not LS_MEin and AN_MEin) --> not AN_MEin
```

This property means when a failure occurs in the hardware (*LS_MEin=false*), the output variable *AN_MEin* of the LBG automaton will eventually be false. This property is valid and the proof has been completed within seconds.

4.3.4 Liveness of the data-flow design. The second model could not be proven correct w.r.t. the liveness property with the full configuration (2 Route+2 LBG). Its data-flow oriented design does not allow us to apply state and transition based abstraction techniques. However, symmetry reduction is still possible. A model with reduced configuration (1 Route+1 LBG) was found to be valid for 4 cycles within 130 seconds.

In contrast to the first model, in the case of the second model the liveness property could be proven to be valid for a reduced model without decomposing it. As a possible reason we consider the direct connection between LBG hardware elements and

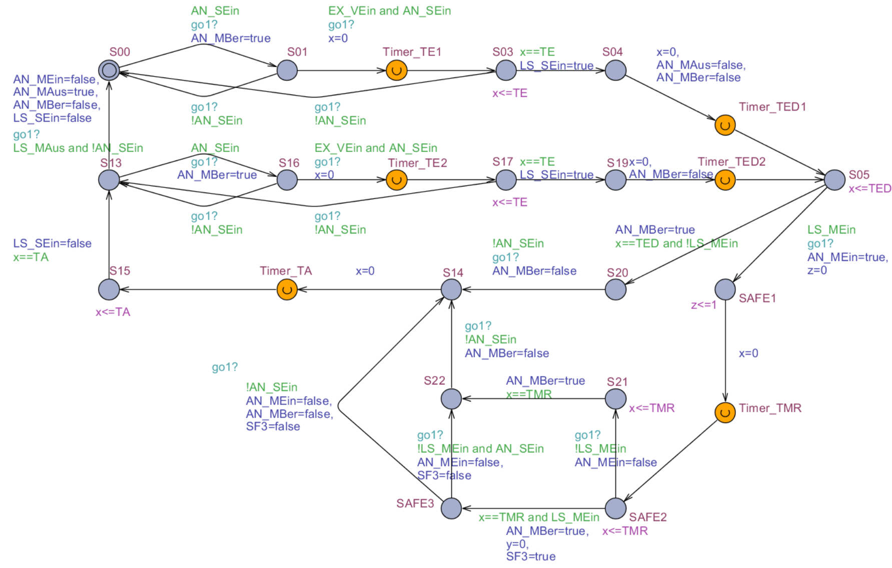

Fig. 8. LBG automaton in UPPAAL

the *LC_Route* operator. In this model, whenever a failure occurs, it is directly sensed by the route controller that controls the monitoring signal. This connection eliminates the effects of the other operators on the liveness property. Hence, this property is not dependent on the feedbacks from the LC_LS_Group operator.

5 Lessons Learned and Conclusions

We performed a comparative case study on formal safety analysis and verification. Our starting point were two functional equivalent SCADE models of a level crossing controller software, both developed in industry. As a formally founded, model-based approach was used in the design already, a seamless expansion towards formal safety analysis and verification within the same tool environment was straightforward. Despite of the difficulties we faced and which fill major parts of our experience report, we were able to identify some of the most relevant critical sets of fault modes as well as to verify numerous safety requirements. In addition, confidence that the results of the formal verification apply for the finally generated executable is strong, since the target code is automatically generated from the SCADE models by the certified code generator. Such a full integration and tool support for design, code generation and verification is mandatory for the successful usage of formal methods in industrial development of safety-critical systems. In this way formal methods can contribute with strong evidence to a safety case a system manufacturer has to provide to certification authorities.

5.1 Lessons Lerned

Our verification results support the hypothesis that applicability of formal analysis based on SAT model checkers depends on the design and modeling style. We found that

the state-based model lends itself better to abstraction and reduction techniques than the data-flow oriented one; in our case study some properties were only provable after reduction and abstraction had been applied. Another argument pro state-based designs is that they facilitate transformation to other model checkers. For data-flow oriented designs some abstraction techniques do not seem obviously applicable.

On the other hand, we found that our liveness analysis could be performed with the data-flow oriented model after a symmetry reduction with the SCADE DV whereas for the state-based model further abstraction and a model transformation to UPPAAL was necessary. In this concrete case this seems to be due to architectural differences. However, from our case study we cannot conclude that data-flow oriented models are in general better suited for formal verification. We faced severe complexity issues during DCCA and liveness analysis for both of our moderately sized models.

After all, we cannot fully explain why the two modeling styles differ w.r.t. verification: The size and complexity of the models are similar. But in the data-flow design the state information is scattered throughout the model and less uniform than in the state-based one. That may be a reason why the model transformation to SCADE DV's SAT-based model checker yields a better result for the state-based model. But to substantiate this assumption the SCADE internal model transformation would have to be inspected in detail.

5.2 Open Issues and Future Work

Next, we discuss a few open issues: Firstly, the heuristics we applied in order to deal with complexity issues during our liveness analysis also could be applied to the formal verifications that happen during the DCCA. Secondly, our model transformation to UPPAAL makes heavy use of the specific form of the given SCADE model (flat state machine, timers). If one aims at an extension to safe state machines *with* hierarchy it is questionable to which extent the flattening that has to be part of any (automatic) transformation will blow-up the resulting timed automata and whether formal verification in UPPAAL will perform well on them. Perhaps, it is possible to exploit compositional techniques in this case, and we leave this as an open question. Thirdly, we believe that the steps we performed in our case study are successfully applicable more generally provided that given SCADE models adhere to similar restrictions. Lastly, we verified our model transformation with the help of testing and did not provide of formal proof of the semantic correctness of our model transformation. Such a correctness proof would, of course, be desirable, but here we leave this for future work.

Finally, we have to state that formal verification is still constrained by scalability problems that hampers its usage in practice. We solved them for the state-based model by transforming it into UPPAAL timed automata, a formalism that offers a much more efficient handling of time. From a methodological point of view, this model transformation is a *time* abstraction. However, it cannot easily be integrated with the synchronous modeling paradigm SCADE is based on. In addition, thorough expertise on verification techniques, the underlying semantics and algorithmics is a pre-requisite to come up with an alternative formalism that potentially can efficiently solve a specific verification problem. However, a desirable improvement of todays verification engines is feedback that, in case of complexity problems, directs the developer to the origin of the

problems within the models. For example, a measure of the impact of selectable model elements or system components on state space size or other complexity measures for the verification problem will be of great help, as it will ease the choice for a promising abstraction or reduction heuristic.

Acknowledgements. We are grateful to Siemens AG, I MO RA, for providing the real industrial models for our formal analysis and to Dr. Dirk Peter for fruitful discussions.

References

1. Abdulla, P.A., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing Safe, Reliable Systems Using Scade. In: Margaria, T., Steffen, B. (eds.) ISoLA. Lecture Notes Comput. Sci., vol. 4313, pp. 115–129. Springer (2004), http://dx.doi.org/10.1007/11925040_8
2. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Information and Computation 104(1), 2–34 (1993)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
4. André, C.: Semantics of S.S.M (safe state machine). Tech. Rep. UMR 6070, I3S Laboratory, University of Nice-Sophia Antipolis (2003)
5. Bozzano, M., Villafiorita, A.: Improving system reliability via model checking: The fsap/nusmv-sa safety analysis platform (2003)
6. CENELEC: EN 50128 – Railway Applications – Software for Railway Control and Protection Systems. European Standard. (2001)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
8. Daskaya, I.: Comparative Safety Analysis and Verification for Level Crossings. Master's thesis, Technische Universität Braunschweig (2011)
9. DIN: EN 50126: Spezifikation und Nachweis der Zuverlässigkeit, Verfügbarkeit, Instandhaltbarkeit und Sicherheit (RAMS) (1999)
10. DIN: EN 50129: Bahnanwendungen – Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Sicherheitsrelevante elektronische Systeme für Signaltechnik (2003)
11. Güdemann, M., Ortmeier, F., Reif, W.: Using deductive cause-consequence analysis (DCCA) with SCADE. In: Proc. 26th Intern. Conference on Computer Safety, Reliability and Security (SAFECOMP). Lecture Notes Comput. Sci., vol. 4680, pp. 465–478. Springer (2007)
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proceedings of the IEEE. vol. 79:9, pp. 1305–1320 (1991)
13. Hanisch, H.M., Pannier, T., Peter, D., Roch, S., Starke, P.: Modeling and formal verification of a modular level-crossing controller design (2000)
14. IEC 60812: Analysis techniques for system reliability (2006)
15. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements (1998), corrigendum 1999
16. Joshi, A., Whalen, M.: Modelbased safety analysis: Final report. Tech. rep., NASA (2005)
17. Lamport, L.: What good is temporal logic. Information Processing 83, 657–668 (1983)
18. McDermid, J.A., Nicholson, M., Pumfrey, D.J., Fenelon, P.: Experience with the application of HAZOP to computer-based systems. In: 10th Annual Conference on Computer Assurance (Compass). pp. 37–48 (1995)
19. Ortmeier, F., Reif, W., Schellhorn, G.: Deductive cause consequence analysis (DCCA). In: Proc. IFAC World Congress. Elsevier, Amsterdam (2006)
20. UPPAAL 4.0: Small Tutorial. http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf (2009), November 16, 2009